Asignatura:
66.20 Organización de computadoras

Profesor: Hamkalo, José Luis

Trabajo práctico 3: Datapath y pipeline

Nombre y apellido	Padrón	Correo electrónico	Slack
Julian Mejliker	100866	jmejliker@gmail.com	Julian Mejliker
Joel Nicolas Saidman	99730	Joelsaidman1@gmail.com	Joel Saidman
Luciano Sportelli Castro	99565	sportelliluciano@gmail.com	Lucho Sportelli Castro

Primer cuatrimestre 2019

Índice

1. Introducción

El objetivo del trabajo práctico es interiorizarse con las arquitecturas de computadoras segmentadas, utilizando de forma práctica una arquitectura tipo MIPS32 simulada.

Durante el desarrollo del trabajo se realizarán modificaciones a implemetaciones ya provistas de la arquitectura MIPS en versión uniciclo y multiciclo. Las modificaciones a realizar consisten en el agregado de ciertas instrucciones a la arquitectura, tales como saltos y *shifts* teniendo en consideración los riesgos que pueden producirse en el caso multiciclo.

La arquitectura uniciclo ejecuta las instrucciones de a una por vez, tomando varios ciclos de reloj para completar una instrucción; mientras que la arquitectura multiciclo está segmentada en cinco etapas, manteniendo la cantidad de ciclos de reloj requeridos por instrucción pero ejecutando más de una instrucción al mismo tiempo, y por consiguiente, obteniendo un CPI más bajo.

2. Implementar `j` en multiciclo

En este ejercicio se pide implementar la instrucción `j` en la arquitectura multiciclo.

Para esto se decidió utilizar la estructura ya implementada para las instrucciones de tipo *branch*, aprovechando que las instrucciones de tipo *jump* cumplen el mismo objetivo: alterar el valor del *Program Counter*. La diferencia entre las mismas radica en el direccionamiento. Las instrucciones *branch* direccionan de forma relativa a la posición actual mientras que las de tipo *jump* lo hacen de forma absoluta.

Analizando el comportamiento de las instrucciones de tipo *branch* se puede observar que durante la etapa *instruction fetch* (de acá en adelante, `IF`) se toma el valor inmediato de la instrucción, se le extiende el signo y se lo propaga a la siguiente etapa. A sí mismo se propaga directamente el valor del `PC` que tomará la siguiente instrucción a la próxima etapa para poder hacer el direccionamiento relativo al mismo.

Para implementar la instrucción `j` se reemplaza el valor del `PC` propagado a la siguiente etapa por la dirección de destino del salto. Esto requiere forjar en la etapa `IF` el valor absoluto de la dirección, lo cual se hace del mismo modo que está implementado en el caso uniciclo: un *shift* para agregar los dos bits menos significativos, un distribuidor y un concatenador para los cuatro más significativos. Como observación, podría reemplazarse el *shift* por una concatenación con una constante (ya que siempre los dos bits menos significativos serán 00).

Esta implementación presenta un inconveniente en la etapa *instruction decode*, (en adelante `ID`) ya que para las instrucciones de tipo *branch* el direccionamiento es relativo a la posición previa del `PC`. El valor sumado en esta etapa es parte del inmediato de la instrucción, que en este caso está ocupado por un fragmento de la dirección absoluta, es decir, en el caso común es un valor distinto de 0, que terminaría alterando el valor correcto ya calculado en la etapa anterior. Este inconveniente fue solucionado utilizando un multiplexor que provee al sumador el valor inmediato en el caso de que se trate de un *branch* o un 0 en el caso de un *jump*.

Dado que el control del flujo de ejecución no fue modificado, no hay riesgos posibles, ya que la implementación provista ya se encarga de vaciar el *pipeline* al ejecutar el salto.

A la unidad de control sólo se le agregó una nueva señal que toma el valor alto cuando la instrucción es un salto y bajo en caso contrario.

En la siguiente figura se pueden observar las modificaciones realizadas:

- Dos multiplexores, el que se encuentra en la etapa `IF` para determinar cual va a ser el valor nuevo del `PC` a pasar a la siguiente etapa; y el que se encuentra en la etapa `ID` para determinar

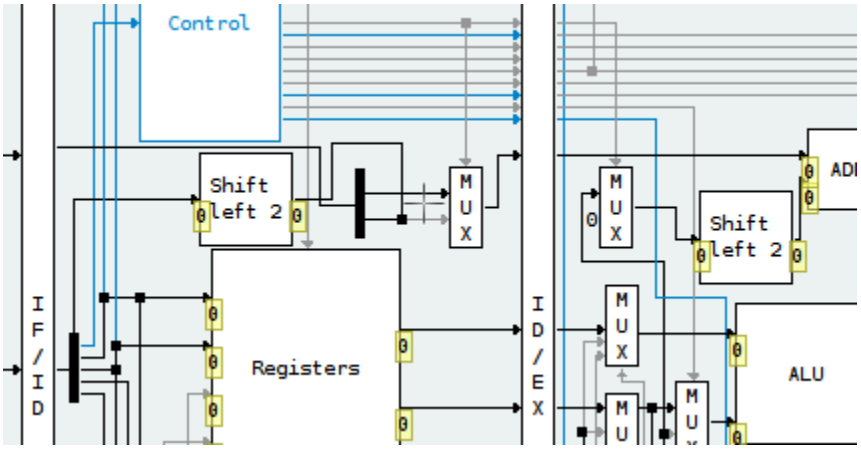

Figura 1: Modificación realizada al *pipeline* para la implementación de la instrucción `j`

si se debe sumar el offset del salto o no.

- Un *shift* para agregar los dos ceros al valor absoluto del destino del salto.
- Un concatenador para agregar los 4 bits más significativos del `PC` al valor absoluto del destino del salto.
- Se agregó una nueva señal de control, que controla ambos multiplexores agregados. Para esto se conecta directamente al multiplexor que está en la etapa `IF` y a través del registro de *pipeline* al multiplexor que está en la etapa `ID`.

2.1. Código de pruebas

Se corrió el siguiente código de pruebas para verificar el correcto funcionamiento:

```
addi $t0, $0, 5
j label
addi $t0, $0, 10
nop
nop
label:
    addi $t1, $0, 7
```

Se comprueba que al finalizar el programa los registros quedan con los valores correctos.

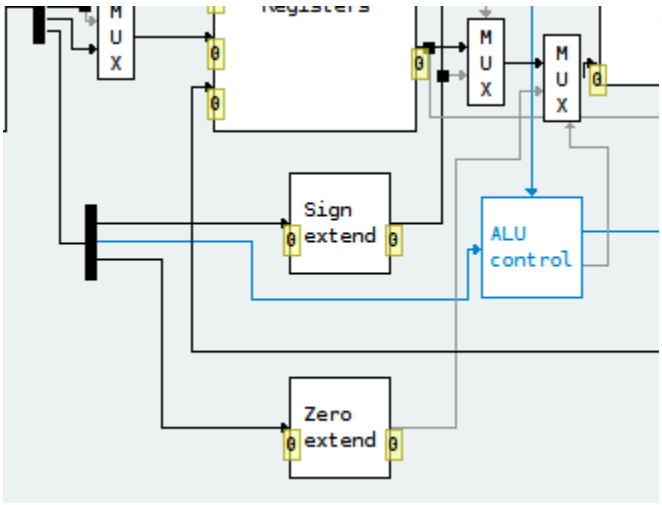

Figura 2: Modificaciones realizadas al *datapath* uniciclo para agregar la instrucción `sll`

3. Implementar `sll` en uniciclo

Se pide implementar la instrucción `sll`, *Shift Left Logical* en el `datapath` uniciclo.

Para esto se debe decodificar el valor *shamt* (*Shift Amount* que viene embebido en la instrucción y agregar a la ALU y a su unidad de control la operación `sll`.

La implementación realizada sólo agrega un extensor con ceros para extender el valor embebido en la instrucción de 5 a 32 bits y un multiplexor para determinar si el valor a utilizar como segundo operando de la ALU proviene desde este extensor o desde el camino normal (registro o inmediato). A sí mismo se le agregó una nueva señal de salida a la unidad de control de la ALU para que control este último multiplexor. La señal se pone en alto si la operación a ejecutar en la ALU es un *shift*.

Se pueden observar las modificaciones en la siguiente figura:

3.1. Código de pruebas

Se utilizó el siguiente código de pruebas:

```
addi $t0, $0, 7
sll $t0, $t0, 17
```

Se comprobó que los registros quedan con los resultados esperados.

4. Implementar `srl` en multiciclo

Se pide implementar la instrucción `srl` (*Shift Right Logical* en el *datapath* multiciclo.

Se procede en este caso de forma similar al anterior, teniendo en cuenta las diferencias debido a estar segmentado. La implementación realizada extiende el valor de 5 a 32 bits en la etapa `IF`. Realizar esta operación en esta etapa se realiza únicamente por simplicidad a la hora de ubicar los componentes en el diagrama, ya que requiere pasar 32 bits de `IF` a `ID`, cuando se podría únicamente pasar 5 bits y realizar la extensión directamente dentro de `ID`.

En la etapa `ID` se agrega una entrada nueva al multiplexor que determina cuál es el segundo operando de la ALU, y se le agrega una nueva señal de control que proviene del control de la ALU del mismo modo que en el caso anterior.

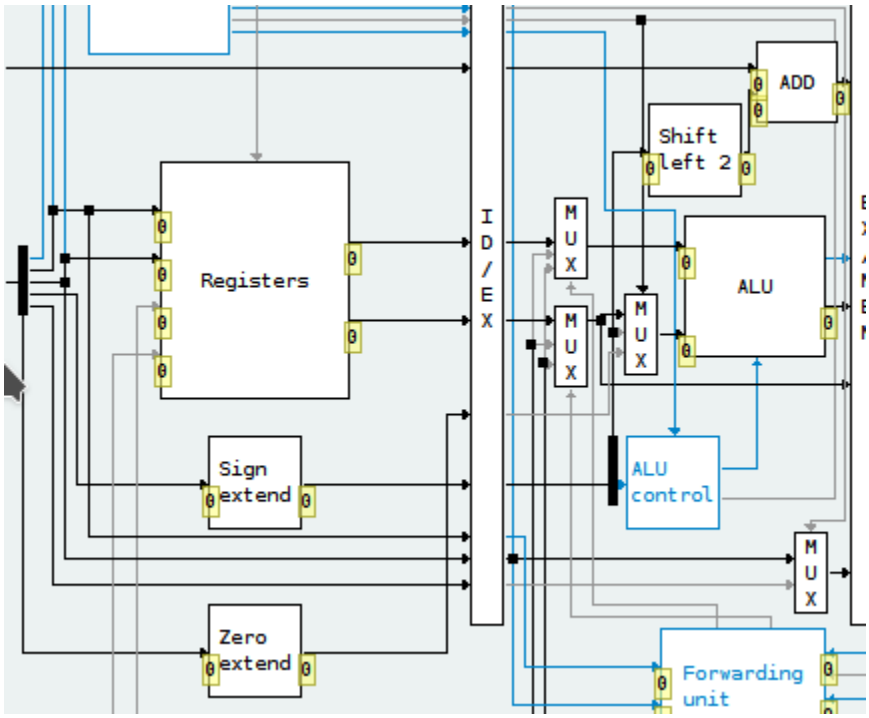

Figura 3: Cambios realizados en el *datapath* multiciclo para incorporar la instrucción `srl`

4.1. Código de pruebas

Se utilizó el siguiente código de pruebas:

```
addi $t0, $0, 7
addi $t1, $0, 17
loop:
    beq $0, $t1, done
    add $t0, $t0, $t0
    addi $t1, $t1, -1
    b loop
done:
srl $t1, $t0, 17
```

Se observa el correcto valor de los registros al finalizar la ejecución.

5. Implementar `jalr` en uniciclo

Se pide implementar la instrucción `jalr` (*Jump And Link Register)* en el *datapath* uniciclo.

Se observa según el manual de la arquitectura que esta instrucción utiliza el formato de las aritméticas.

Esta instrucción requiere cambios en la forma en que se actualiza el `PC` y en la forma en que se almacenan los datos en registros ya que la información a almacenar proviene ahora del `PC`.

Para esto se agregó una nueva entrada al multiplexor que controla qué valor se escribe en el `PC` cuyo valor proviene del primer registro que se envía a la ALU, y una nueva entrada al multiplexor que controla qué dato se escribe en el registro de destino, cuyo valor proviene del multiplexor que indica `PC+4` o la dirección de un *branch*, aprovechando que sabemos de antemano que siempre que se ejecute esta instrucción estará en `PC+4`. Ambos multiplexores están comandados por la señal original en el bit menos significativo y por una nueva señal de control agregada a la unidad de control de la ALU que indica si se trata de la instrucción en cuestión en el bit más significativo.

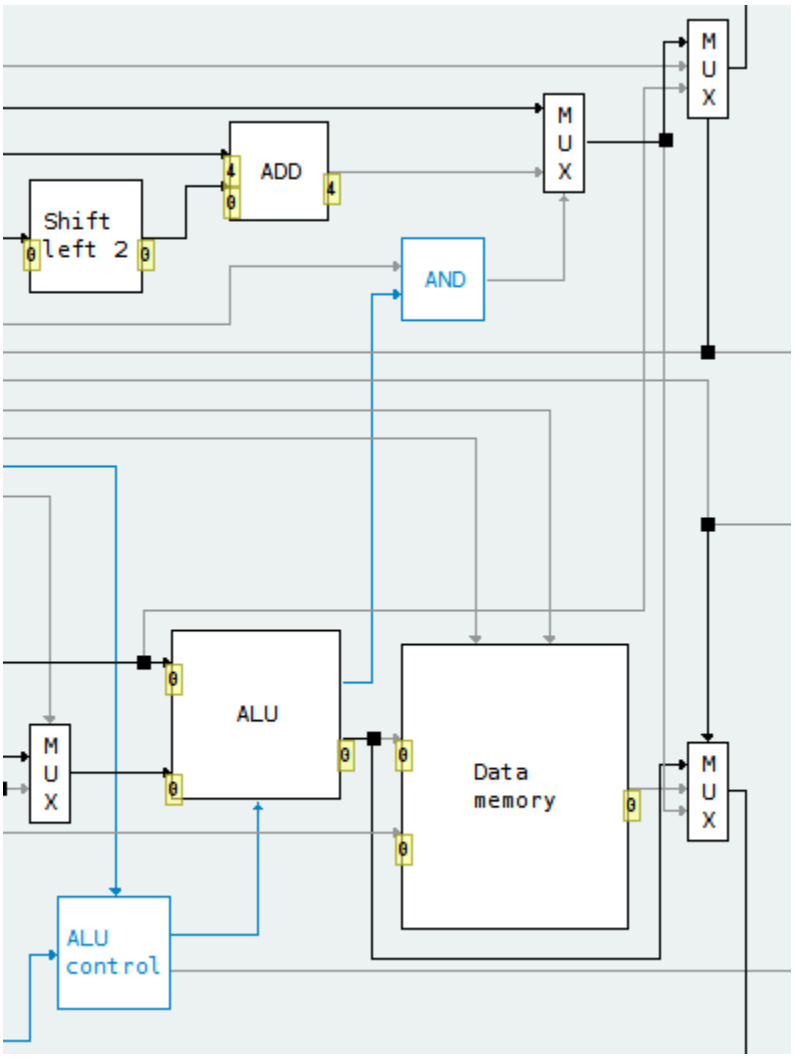

Figura 4: Modificación del *datapath* uniciclo para incorporar la instrucción `jalr`

5.1. Código de pruebas

Se utilizó el siguiente código de pruebas:

```
addi $t0, $0, label
jalr $t0
addi $t3, $0, 7
nop
nop
label:
    addi $t1, $0, 10
    jalr $ra
```

Se comprobó el correcto funcionamiento y los valores finales de los registros.

6. Implementación de `jalr` en multiciclo

Se pide implementar la misma instrucción que en el caso anterior, pero en el *datapath* multiciclo.

Nuevamente se aprovecha el funcionamiento ya existente para las instrucciones de tipo *branch*. En este caso, el multiplexor que determina cuál es la siguiente instrucción toma su segundo valor desde el registro fuente 1.

En este caso es necesario además propagar el valor de la próxima instrucción a ejecutar si no hubiera salto hasta la última etapa ya que se debe almacenar su valor en el registro `$ra`.

Los cambios realizados en la etapa `ID` para ejecutar el salto son idénticos al caso de la instrucción `j`.

Para poder almacenar el valor de la próxima instrucción a ejecutar se propaga la señal de control `Jump`, agregada en el primer ejercicio hasta la última etapa, junto con el valor `PC+4`. En la última etapa se agrega una nueva entrada al multiplexor que determina si se escribe desde registros o desde memoria para indicar la nueva fuente de datos que será el valor propagado. A éste último multiplexor se le agrega la señal de control propagada `Jump` en su bit de selección más significativo para almacenar en el registro correspondiente el valor propagado.

Debido a haber aprovechado la lógica ya implementada para las instrucciones de tipo *branch* no hay riesgos posibles, ya que el mismo sistema de control se encarga de vaciar el *pipeline* al ejecutar la instrucción.

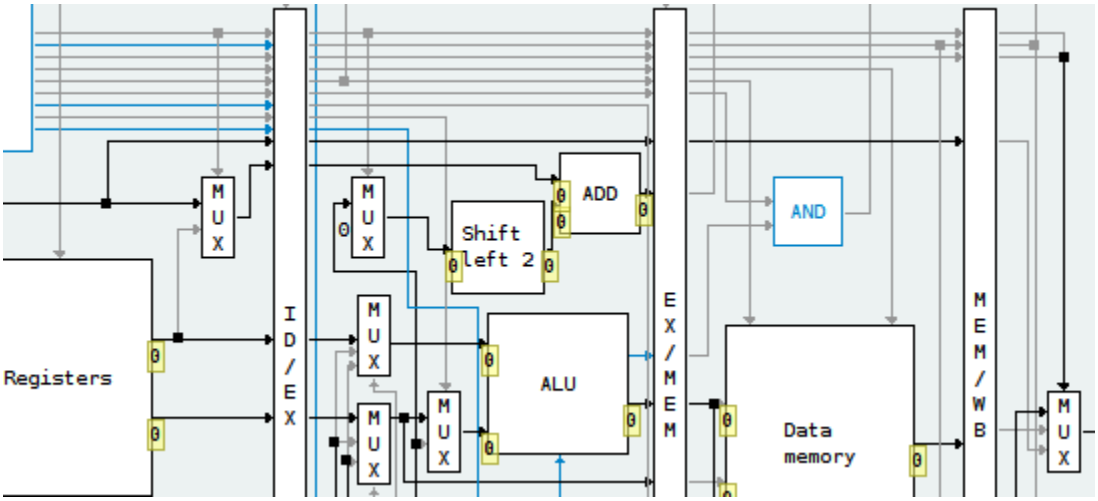

Figura 5: Modificaciones realizadas al *datapath* multiciclo para incorporar la instrucción `jalr`

6.1. Código de pruebas

Se utilizó el mismo código de pruebas que en el caso uniciclo. Se comprobó el correcto funcionamiento y los valores finales de los registros.

7. Conclusión

Se puede observar una misma instrucción puede implementarse de distintas formas obteniendo el mismo resultado, pero con distintos costos en cantidad de hardware necesario y latencias. En la implementación de la instrucción `srl`, por ejemplo, se puede observar que nuestra implementación

realiza la extensión de signo en la etapa `IF` requiriendo así que se propaguen 32 líneas de una etapa a la siguiente, aún cuando 27 de las líneas contienen 0. Dicha implementación funcionaría de la misma manera pero con un costo menor si la extensión de signo se realizara en `ID`, ahorrándose estas 27 líneas. En este caso la razón para realizarlo de esta manera es meramente para obtener un diagrama más claro en el simulador; debería determinarse en el momento de construirlo sobre hardware real si tiene sentido agregar más líneas de una etapa a otra o tener una de las etapas mucho más compleja que el resto.

Por otra parte se observa que la implementación de instrucciones en un *datapath* segmentado es ligeramente más difícil que en uno uniciclo ya que debe tenerse en cuenta la posibilidad de presentarse riesgos de dependencias entre datos o debido a instrucciones de control de flujo como saltos.